Kontrola pristupa koristeći informacije iz kreditnih kartica

Nakić, Mate

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Mate Nakić

KONTROLA PRISTUPA KORISTEĆI INFORMACIJE IZ KREDITNIH KARTICA

ZAVRŠNI RAD

Varaždin, 2021.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mate Nakić

Matični broj: 44880/16-R

Studij: Informacijski sustavi

KONTROLA PRISTUPA KORISTEĆI INFORMACIJE IZ KREDITNIH KARTICA

ZAVRŠNI RAD

Mentor :

doc. dr. sc. Boris Tomaš

Varaždin, kolovoz 2021.

Mate Nakić

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Tema završnog rada je izrada digitaliziranog sustava za kontrolu pristupa upotrebom kreditne kartice. Cilj rada je autorizirati pristup pomoću kreditne kartice ali na način da se ne naruši privatnost te da osobni podaci ne dolaze u doticaj sa sustavom. Uvodno se spominje tradicionalna ručna kontrola pristupa i problemi iste kao i hipoteze i ciljevi rada. Opisana je izrada aplikacije na web platformi sačinjena od infrastrukture, pozadinske aplikacije i grafičkog sučelja. Opisana je izrada sefa (komponente i upravljački softver) i integracija s web aplikacijom te način na koji se autoriziranje obavlja bez narušavanja privatnosti. Rad je zaključen s opisom digitalnog rješenja kao odgovor na probleme ručne kontrole pristupa.

Ključne riječi: web servis; arduino; iot; nfc; esp32

Sadržaj

1. Uvod

Svakim danom svijet se sve više digitalizira i mehanički procesi se automatiziraju kako bi se uštedjelo na vremenu i resursima. Sve više smo svjesni koliko računala lako i s manje grešaka obavljaju ponavljajuće zadatke, za razliku od ljudi. Jedan od ponavljajućih i sklon pogrešci zadatak jest kontrola i evidencija pristupa. Proces zahtjeva dva sudionika: kontrolora i korisnika. U tradicionalnom (ne-digitaliziranom) okruženju kontrolor je osoba koja prima zahtjeve korisnika, prema određenim uvjetima dozvoljava ili odbija pristup korisniku te evidentira u evidencijsku knjigu komu i kad je pristup dozvoljen ili odbijen. Proces je vrlo jednostavan, no ima više prostora za pogrešku:

- Je li kontrolor prisutan?
- Jesu li uvjeti stvarno ispunjeni (zloupotreba ovlasti ili ljudska greška)?
- Jesu li podaci korisnika ispravni (npr. ime i prezime)?
- Jesu li meta podaci ispravni (npr. datum i vrijeme)?

Digitalizacijom ovog procesa kontrolor više nije osoba već uređaj, evidencijska knjiga nije list papira nego oblak (eng. *cloud*). Ista pitanja mogu biti postavljena nakon digitalizacije, no vjerojatnost pogreške je znatno manja.

Praktični dio rada je implementacija kontrole pristupa (eng. *access control*) pomoću potrošačkih elektroničkih komponenti (eng. *consumer electronics*), oblaka i kreditnih kartica. Elektroničke komponente tvore uređaj (kontrolor) u obliku sefa s kojim korisnik komunicira dok kreditna kartica služi kao sredstvo identifikacije. U oblaku se nalazi aplikacija u kojoj se definiraju kreditne kartice koje imaju pravo pristupa. Uspješni i neuspješni pokušaji pristupanju se evidentiraju u bazu podataka u oblaku.

1.1. Hipoteze i ciljevi

Sljedeće su hipoteze:

- **H1**: Moguće je odobriti pristup sefu pomoću jedinstvenog identifikator kreditne kartice bez narušavanja privatnosti (čitanje osobnih podataka).

Ciljevi rada su:

- **C1**: Razviti pločicu s mikrokontrolerom i potrebnim perifernim komponentama i smjestiti ih u sef.
- **C2**: Omogućiti konfiguriranje bežične pristupne točke sefa.
- **C3**: Izraditi aplikaciju za kontrolu pristupa sefu.

2. Korišteni alati i tehnologije

Prilikom izrade web aplikacije korišteni su sljedeći programski jezici, radni okviri, biblioteke, alati i platforme:

- PHP 8.0 - skriptni jezik korišten za izradu pozadinske aplikacije
- Laravel 8.54 - radnik okvir za PHP aplikacije
- PhpStorm - integrirano razvojno okruženje za izradu PHP aplikacija
- HTML, CSS, JavaScript - standardni jezici za izradu grafičkog sučelja na web tehnologijama
- Vue 2.6.21 - radni okvir za visoko interaktivna grafička web sučelja
- WebStorm - integrirano razvojno okruženje za izradu aplikacija temeljenih na web tehnologijama
- Kubernetes 1.21.2 - platforma za objavljivanje i ažuriranje web aplikacija
- DigitalOcean - platforma za stvaranje resursa u oblaku
- Pulumi 3.11.0 - alat za definiranje i objavu Kubernetes resursa i resursa u oblaku

Za izradu sefa i pripadnoga upravljačkog programa korišteni su sljedeći programski jezici, radni okviri, biblioteke, alati i platforme:

- C++ 11 - jezik opće namjene korišten za izradu upravljačkog programa
- Arudino - radni okvir namijenjen *embedded* programiranju
- PlatformIO 5.1.1 - platforma za kompiliranje i prijenos upravljačkog programa na ESP32 mikrokontroler
- MRFC522 1.4.8 - biblioteka za rad s MFRC522 čitačem kartica
- WiFiManager 0.16 - biblioteka za konfiguriranje pristupne točke
- CLion - integrirano razvojno okruženje za izradu aplikacija u C i C++ jeziku
- bušilica, lemilica i slični alati potrebni za integraciju elektroničkih komponenti u sef

Ovaj dokument pisan je *LaTeX* jezikom, a kompiliran je u PDF format pomoću *latexmk* programa. Izvorni kod web aplikacije, upravljačkog programa kao i ovog dokumenta verzioniran je s *git* alatom na *GitHub* platformi.

3. Web aplikacija

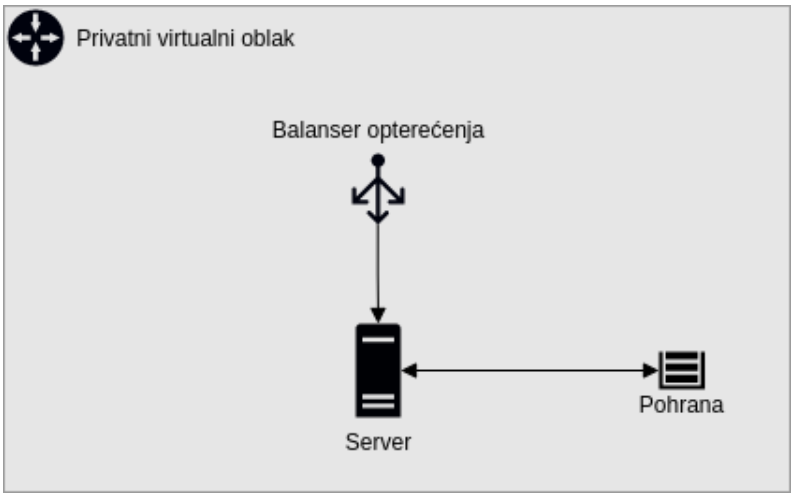
Sustav kontrole i evidencije pristupa viđen s visoke razine sastoji se od tri cjeline:

- Infrastruktura
- Pozadinska aplikacija
- Grafičko sučelje

U ovom poglavlju sve tri cjeline će biti opisane i razrađene.

3.1. Infrastruktura

Infrastruktura je sačinjena od servera, čvorova, usmjerivača i sl. Također, infrastrukturu čini operacijski sustav, vatrozid, i ostali softver[1]. Po tom pitanju danas se ništa značajno nije promijenilo, samo su fizičke komponente skrivene u "oblaku" (eng. *cloud*). Rad zahtjeva vrlo jednostavnu infrastrukturu pa je oblak prikladno rješenje jer ne zahtjeva vlastoručnu konfiguraciju i održavanje fizičkih komponenti.



Slika 1: Fizičke komponente infrastrukture

Na slici 1 nalaze se sve komponente kreirane kod pružatelja usluge oblaka (DigitalOcean). Privatni virtualni oblak ima mnoštvo značajki, ali u ovom slučaju se koristi samo za logičko grupiranje komponenti. Balanser opterećenja (eng. *load balancer*) služi kao *proxy* do glavnog servera i moguće je povezati domenu na njega. Server ima vlastitu sekundarnu pohranu na kojoj se nalazi operacijski sustav i potrebni alati ali i vanjsku pohranu na kojoj se nalaze aplikacijski podaci.

Vanjska pohrana i balanser opterećenja izgledaju, na prvi pogled, nepotrebni. U scenariju kad se glavni server nepovratno sruši vrlo je lako kreirati novi i nastaviti s normalnim radom. Krajnji korisnik osjeti kratki prekid usluge, a ne potpuni gubitak usluge i podataka. U slučaju samo jednog servera (bez vanjske pohrane i balansera opterećenja) podaci i IP adresa se trajno gube što predstavlja veliku neugodnost krajnjem korisniku.

3.1.1. Infrastruktura aplikacije

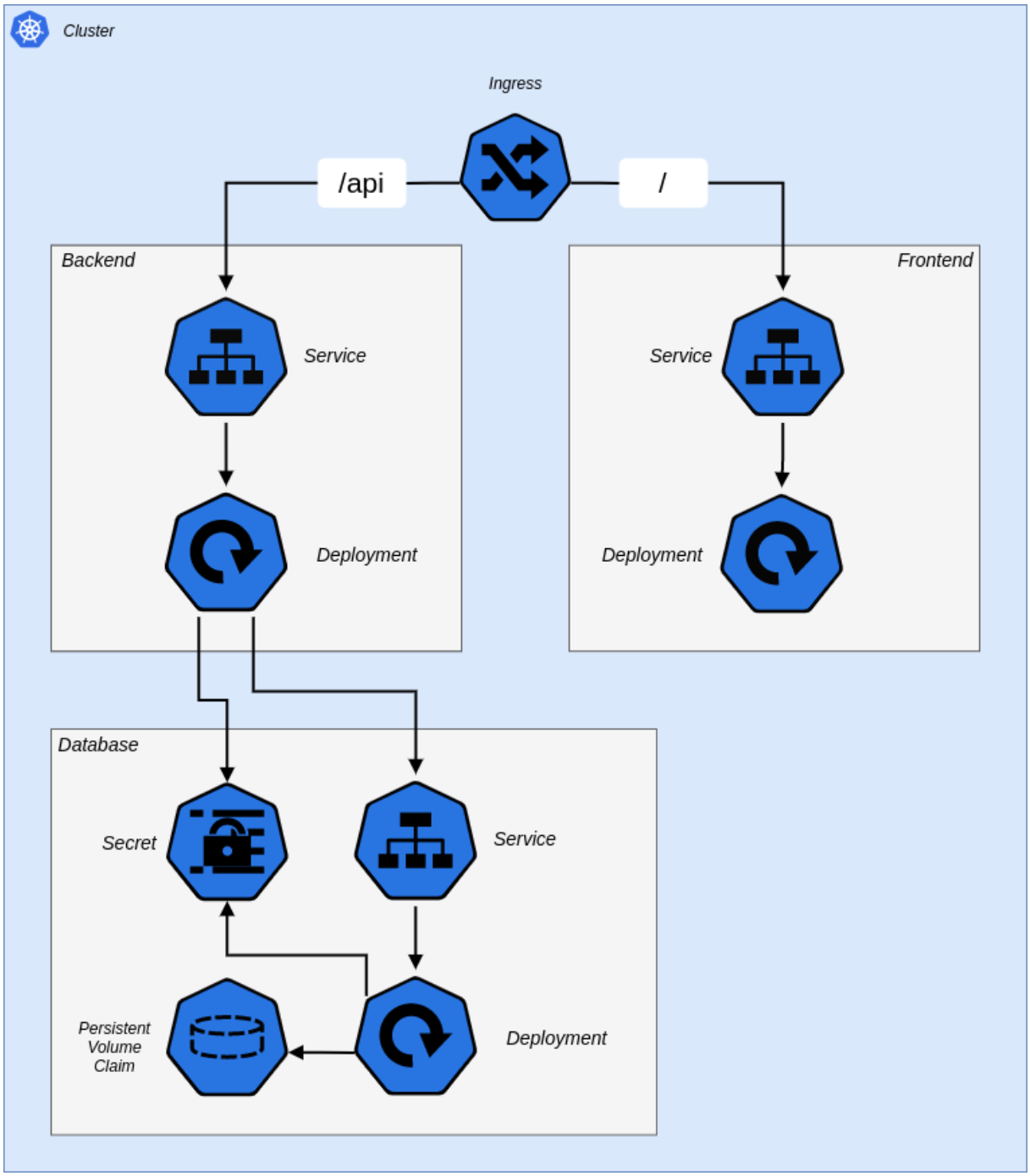
Moderne aplikacije zahtijevaju različite servise i alate kako bi funkcionirale u potpunosti. Baza podataka, red čekanja, posrednik poruka i web server samo su neki od servisa. Potrebni su alati za ažuriranje, objavljivanje i nadzor aplikacija i servisa.

Danas je najpopularniji način objavljivanja aplikacija pomoću kontejnera. Kontejner je rezultat procesa pakiranja aplikacijskog koda sa svim potrebnim bibliotekama i alatima kako bi ga se moglo lako i pouzdano prenositi između okruženja, npr. lokalno (na računalu programera), testno (*staging*) i stvarno (*production*) [2].

Sam kontejner nema nikakvu funkciju dok se ne pokrene. Kontejneri nisu izvršne datoteke (*.exe*) da ih operacijski sustav može pokrenuti bez dodatnih alata. Čak i kad bi operacijski sustav to mogao, kontejner je moguće zaustaviti, ponovno pokrenuti, može završiti u stanju greške (eng. *error state*) te je potrebna "logika" kad i kako treba poduzeti određene akcije. Operacijski sustav se bavi puno nižim stvarima pa problem upravljanja kontejnerom izlazi iz domene operacijskog sustava.

Aplikacije su uglavnom sačinjene od više modula i servisa pa je potrebno i više kontejnera. Ručno upravljanje s istima je vremenski i resursno zahtjevan proces, ali postoje alati koji olakšavaju upravljanje kontejnerima. To su alati koji pomažu pri orkestriranju kontejnera (eng. *container orchestration*) i jedan od njih je i *Kubernetes*. Kubernetes je sustav koji pomaže pri automatiziranju procesa objave i ažuriranja, skaliranja i sveukupnog upravljanja aplikacijama i servisima [3].

Razmjer praktičnog rada ne zahtjeva mnoštvo značajki koje Kubernetes nudi, samo osnovnu značajku: grupira kontejnere koji čine aplikaciju u logičke jedinice [3]. To omogućava lakše razumijevanje komponenti sustava kao i vizualizaciju istih.



Slika 2: Servisi i komponente sustava

Na slici 2 je prikazan *cluster* sa svim servisima i komponentama. Početna točka je *ingress* koji funkcionira kao *proxy*. Ovisno o putanji koju korisnik zahtjeva, zahtjev se prosljeđuje na *backend* (pozadinska aplikacija) ili *frontend* (grafičko sučelje).

Grafičko sučelje se sastoji od dvije komponente: *Service* i *Deployment*. *Deployment* zna kako pokrenuti kontejner, kad je kontejner spreman za obradu zahtjeva i koliko replika ima. Kako svaki kontejner ima svoju IP adresu ostali servisi bi morali znati točnu IP adresu kontejnera kako bi ga mogli koristiti. *Service* predstavlja apstrakciju nad kontejnerima i ponaša se kao balanser opterećenja (u slučaju više replika) ili *proxy* (u slučaju jedne replike).

Pozadinska aplikacija je slična grafičkom sučelju osim što ovisi o bazi podataka. Ovisi o *Service* komponenti baze podataka kao i pristupnim podacima koji se nalaze u komponenti *Secret*. Kako bi pozadinska aplikacija komunicirala s bazom podataka potrebno je nekoliko podataka:

- Adresa
- Korisničko ime
- Lozinka
- Naziv baze podataka

Kako je već rečeno da je *Service* apstrakcija nad kontejnerima tako je sam naziv servisa ujedno i **adresa**. **Korisničko ime**, **lozinka** i **naziv baze podataka** se nalaze u *Secret* komponenti. Ona predstavlja jedini izvor točnosti (eng. *single source of truth*) pa ukoliko dođe do promjena u podacima pozadinska aplikacija i baza podataka mogu pravilno reagirati kako bi nesmetano funkcionirali.

Baza podataka, naspram pozadinske aplikacije, ima jednu komponentu više: *Persistent Volume Claim*. Svaki kontejner ima sustav datoteka odvojen od domaćinovog sustava datoteka (eng. *host file system*). U slučaju zaustavljanja kontejnera (namjerno ili uslijed greške) svi podaci bi bili trajno izgubljeni. Kako bi se to izbjeglo potrebno je dozvoliti kontejneru pisanje u domaćinov sustav datoteka. Komponenta *Persistent Volume Claim* apstrahira ovu funkcionalnost pa sam kontejner uopće ne zna da se podaci zapravo zapisuju na domaćinov sustav datoteka. To omogućava djelomično fizičko odvajanje domaćinovog sustava datoteka. Direktoriji koji su potrebni bazi podataka su odvojeni na vanjsku pohranu (slika 1) dok su podaci i datoteke o kojima ovisi operacijski sustav pohranjeni na internu pohranu servera. U slučaju nepovratnog pada servera (npr. fizičko oštećenje) zamjenom servera sustav nastavlja raditi bez gubitka podataka.

3.2. Pozadinska aplikacija

Pozadinska aplikacija definira većinski dio logike sustava. Svoje funkcionalnosti daje na korištenje kroz REST API kako bi grafičko sučelje i sef mogli ispunjavati svoje uloge. Funkcionalnosti su sljedeće:

- Upravljanje karticama
- Autorizacija kartica
- Pregled dnevnika pristupa
- Autorizacija korisnika

Funkcionalnosti su implementirane u programskom jeziku *PHP* uz pomoć *Laravel* radnog okvira.

3.2.1. Upravljanje karticama

Entitet "Kartica" sadrži podatke o imenu kartice i jedinstvenom identifikatoru (*UID*). Sadrži i meta podatke o datumu stvaranja i datumu izmjene. Entitetom korisnik upravlja kroz grafičko sučelje, a pozadinska aplikacija koristi jedinstvene identifikatore pri autoriziranju zahtjeva sefa.

Entitet kartice nema relacija na druge entitete pa nema smisla prikazivati ER dijagram već je dovoljan programski kod. Radni okvir omogućava definiranje migracija baze podataka kroz programski jezik umjesto pisanja čistog SQL-a.

```
class CreateCardsTable extends Migration
{
    public function up()
    {
        Schema::create('cards', function (Blueprint $table) {
            $table->id();
            $table->string('name')->unique();
            $table->string('uid')->unique();
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('cards');
    }
}
```

Radni okvir ima ORM (*Object Relational Mapper*) zvan *Eloquent* što omogućava rad s bazom kroz objekte umjesto ručnog pisanja SQL upita. Dovoljno je definirati klasu prema imenu tablice, u jednini, i naslijediti klasu *Model* iz radnog okvira.

```
class Card extends Model
{
    static function existsByUid(string $uid)
    {
        return static::query()->where('uid', '=', $uid)->exists() === true;
    }
}
```

Statična metoda *existsByUid* provjerava postoji li zapis kartice s danim *UID-em* i vraća *true* ili *false*. Ona je osnovica za autorizacijske zahtjeve sefa.

Operacije nad entitetom kartice su izložene (dane na korištenje) vanjskom svijetu, odnosno grafičkom sučelju kroz niz HTTP putanji:

Tablica 1: Prikaz putanja svih funkcionalnosti upravljanja karticama

Funkcionalnost	HTTP Metoda + Putanja	Klasa@Metoda
Popis svih kartica	GET /api/card	CardController@index
Stvaranje kartice	POST /api/card	CardController@store
Uređivanje kartice	PUT /api/card/{id}	CardController@update
Brisanje kartice	DELETE /api/card/{id}	CardController@destroy

Iz tablice 1 može se vidjeti da sve funkcionalnosti dijele isti dio putanje URL-a: /**api**/**card** kao i istu klasu u kojoj su implementirane metode: **CardController**. Kako su *Create, Read, Update, Delete (CRUD)* operacije nad istim entitetom ponavljajući uzorak u većini aplikacija, radni okvir omogućava jednostavno definiranje putanja i pripadnih metoda:

```
Route::resource('card', CardController::class)->only(['index', 'store', 'update', '
    destroy']);
```

Prema tablici 1 popis svih kartica je implementiran u metodi **index**. Jednostavan upit dohvaća osnovne podatke o karticama, sortira ih po datumu stvaranja i vraća u JSON formatu.

```
public function index(): JsonResponse
{
    $cardsByCreationDateAscending = Card::query()->select(['id', 'name', 'uid'])->
        orderBy('created_at')->get();

    return response()->json([
        'data' => $cardsByCreationDateAscending
    ]);
}
```

Prema tablici 1 stvaranje nove kartice je implementirano u metodi **store**. Metoda stvara objekt *Card* i popunjava ga s podacima dobivenim od strane korisnika. Kod uspješnog spremanja vraćaju se svi podaci o kartici u JSON formatu, a kod neuspješnog spremanja vraća se kratka poruka, također u JSON formatu.

```
public function store(StoreCard $request): JsonResponse
{
    $name = $request->name;
    $uid = $request->uid;

    $card = new Card();
    $card->name = $name;
    $card->uid = $uid;

    if ($card->save()) {
        return response()->json([
            'data' => $card->fresh()
        ]);
    }
```

```
    return response()->json(['data' => 'Neuspjesna pohrana kartice. Pokusajte ponovo
        !']);
}
```

Tip argumenta *$request* je *StoreCard*. To je klasa koju radni okvir instancira prilikom obrađivanja zahtjeva i ona služi za validaciju podataka. Ukoliko uvjeti nisu zadovoljeni, metoda *store* se neće izvršiti već se vraća kratka poruka sa svim problemima u JSON formatu.

```
class StoreCard extends FormRequest
{
    public function rules(): array
    {
        return [
            'name' => [
                'required',
                'string',
                'max:30',
                Rule::unique(Card::class, 'name')
            ],
            'uid' => [
                'required',
                'string',
                'max:40',
                Rule::unique(Card::class, 'uid')
            ]
        ];
    }

    public function messages(): array
    {
        return [
            'name.unique' => 'Ime vec postoji.',
            'name.max' => 'Ime moze biti maksimalno 30 znakova.',
            'name.*' => 'Ime je obavezno.',
            'uid.unique' => 'UID vec postoji.',
            'uid.max' => 'UID moze biti maksimalno 40 znakova.',
            'uid.*' => 'UID je obavezan.'
        ];
    }
}
```

Metoda *rules* vraća polje (eng. *array*) svih uvjeta koje treba zadovoljiti. Naziv kartice je obavezan (*'required'*) tekst (*'string'*) do trideset znakova (*'max:30'*) i mora biti unikatan, tj. u bazi podataka ne smije postojati kartica s istim imenom (*Rule::unique*). Slični uvjeti vrijede i za *UID* kartice, razlika je jedino u maksimalnom broju znakova i naziv polja po kojem se gleda uvjet unikatnosti. Metoda *messages* vraća polje s porukama korisnim za korisnika kako bi znao u čemu je problem.

Prema tablici 1 uređivanje postojeće kartice je implementirano u metodi **update**. Funkcionalnost je slična kao i kod stvaranja nove kartice, no implementacija je nešto drukčija.

```
public function update(Card $card, UpdateCard $request): JsonResponse
{
    $card->name = $request->name;

    if ($card->save()) {
        return response()->json([
            'data' => $card->fresh()
        ]);
    }

    return response()->json(['data' => 'Neuspjesno azuriranje kartice. Pokusajte
        ponovo!']);
}
```

Kako se radi o postojećoj kartici potrebno ju je jedinstveno identificirati. Za to se koristi jedinstveni identifikator odnosno primarni ključ kojeg dodjeljuje baza podataka. Pri slanju zahtjeva identifikator je dio putanje (*/api/card/{id}* - tablica 1). Nalazi se u vitičastim zagradama pa radni okvir zna da vrijednost mora proslijediti u pripadajuću metodu (*update*). Zaglavlje metode bi onda izgledalo:

```
public function update(int $id, UpdateCard $request): JsonResponse
```

To implicira manualno slanje upita prema bazi podataka i provjeru postoji li kartica. Kako je ovo česti problem većine aplikacija, radni okvir daje jednostavno rješenje: Tip argumenta postaviti na željeni entitet (klasu) umjesto broja (*int*, *integer*). Radni okvir pretpostavlja da se za uvjet pretrage koristi primarni ključ (*ID*) jer je on jedinstven i ne može biti *null*. Ukoliko entitet postoji, radni okvir prilaže sve informacije o tom entitetu kao argument metode i izvršava metodu. Ukoliko entitet ne postoji, radni okvir zaustavlja izvršavanje skripte i ne poziva metodu.

Validacija podataka zahtjeva drugačije ponašanje.

```
public function rules(): array
{
    $cardUnderModification = $this->route('card');

    return [
        'name' => [
            'required',
            'string',
            'max:30',
            Rule::unique(Card::class, 'name')->ignore($cardUnderModification->id)
        ]
    ];
}
```

Kako korisnik može dati isti podatke (nije ništa promijenio) zahtjev se mora normalno izvršiti, bez ikakvih greški. Sve što je potrebno napraviti je pri provjeri unikatnosti ignorirati karticu koja se uređuje.

Prema tablici 1 brisanje kartice je implementirano u metodi **destroy**. Kao i kod uređivanja kartice, tip argumenta je entitet *Card* kojeg radni okvir automatski ubacuje pri izvršavanju metode. Ovisno o uspješnosti brisanja, prikladna poruka se vraća u JSON formatu.

```
public function destroy(Card $card): JsonResponse
{
    try {
        $card->delete();

        return response()->json(['data' => 'Kartica uspjesno obrisana.']);
    }
    catch (Throwable $exception) {
        return response()->json(['data' => 'Neuspjesno brisanje kartice. Pokusajte
            ponovo!']);
    }
}
```

3.2.2. Autorizacija kartica

Prilikom podnošenja zahtjeva sef mora poslati UID kartice koju treba autorizirati. Ukoliko kartica postoji vraća se HTTP statusni kod 200 kao indikator uspješnog autoriziranja ili 403 kao indikator neuspješnog autoriziranja. U pristupni dnevnik je potrebno stvoriti novi zapis s datumom i UID-em kartice kao i (ne)uspješnost autorizacije.

Autorizacija je izložena na putanji **/api/box/authorize** kroz **POST** metodu,

```
Route::post('/box/authorize', BoxAuthorizationAction::class);
```

a implementirana je u **BoxAuthorizationAction** metodi:

```
class BoxAuthorizationAction
{
    public function __invoke(Request $request): Response
    {
        $uid = (string) $request->uid;

        if (Card::existsByUid($uid)) {
            event(new BoxAccessWasGranted($uid));
            return response()->noContent(200);
        }

        event(new BoxAccessWasProhibited($uid));
        return response()->noContent(403);
    }
}
```

Kako je autorizacijska logika zasebna klasa, koristi se posebna metoda *__invoke* koja omogućuje pozivanje objekta poput običnih funkcija. Prethodno definirana metoda *existsByUid* u entitetu "Kartica" koristi se pri odlučivanju koja akcija se poduzima: odobravanje ili odbijanje zahtjeva. Logika zapisivanja u pristupni dnevnik nije direktno implementirana već se koristi sustav događaja (eng. *event*) i osluškivača (eng. *listener*). Događaji definiraju popratne informacije dok osluškivači koriste te informacije i poduzimaju određene akcije.

Dovoljna su dva događaja:

1. Pristup dozvoljen

```
class BoxAccessWasGranted
{
    private string $uid;

    public function __construct(string $uid)
    {
        $this->uid = $uid;
    }

    public function uid(): string
    {
        return $this->uid;
    }
}
```

2. Pristup odbijen

```
class BoxAccessWasProhibited
{
    private string $uid;

    public function __construct(string $uid)
    {
        $this->uid = $uid;
    }

    public function uid(): string
    {
        return $this->uid;
    }
}
```

Oba događaja primaju *UID* kao parametar i daju ga na raspolaganje osluškivačima kroz metodu *uid()*.

Prije implementiranja osluškivača potrebno je stvoriti entitet "Pristupni dnevnik". On sadrži podatke o kartici (*UID*), datum i vrijeme i (ne)uspješnost pristupa. Poput entiteta "Kartica" i ovaj entitet je vrlo jednostavan pa nema potrebe za ER dijagramom, samo migracijski kod:

```
class CreateAccessLogsTable extends Migration
{
    public function up()
    {
        Schema::create('access_logs', function (Blueprint $table) {
            $table->id();
            $table->string('uid');
            $table->timestamp('at_time');
            $table->boolean('access_granted');
        });
```

```
    }


    public function down()
    {
        Schema::dropIfExists('access_logs');
    }
}
```

Naravno, potrebna je i klasa **AccessLog** koja nasljeđuje klasu *Model* da se izbjegne rad s čistim SQL-om.

```
class AccessLog extends Model
{
    protected $casts = [
        'at_time' => 'datetime',
        'access_granted' => 'boolean'
    ];
}
```

Važno je određena polja prebaciti u određene tipove podataka (atribut **$casts**). Kako baza podataka vraća datum i vrijeme u obliku teksta potrebno je prebaciti tip podataka u klasu koja olakšava rad s datumima: *DateTime*. Polje *at_time* će biti prebačeno u tip *DateTime*, a polje *access_granted* u tip *boolean*.

Nakon definiranja modela moguće je implementirati osluškivače.

```
class LogGrantedBoxAccess
{
    public function handle(BoxAccessWasGranted $event)
    {
        $uid = $event->uid();

        $newLogEntry = new AccessLog();
        $newLogEntry->uid = $uid;
        $newLogEntry->at_time = now();
        $newLogEntry->access_granted = true;

        $newLogEntry->save();
    }
}

class LogProhibitedBoxAccess
{
    public function handle(BoxAccessWasProhibited $event)
    {
        $uid = $event->uid();

        $newLogEntry = new AccessLog();
        $newLogEntry->uid = $uid;
        $newLogEntry->at_time = now();
        $newLogEntry->access_granted = false;

        $newLogEntry->save();
```

```
    }
}
```

Oba osluškivača dijele sličnu logiku: stvore novi *AccessLog* model, zapišu UID iz događaja, datum i vrijeme te (ne)uspješnost pristupa i to pohrane u bazu podataka. U suštini, razlika je u događaju koji osluškuju i vrijednosti polja *access_granted*. Funkcija *now()* vraća objekt tipa *DateTime*, a vrijednost polja *access_granted* je *true* ili *false*. Zbog prethodno definiranog atributa *$casts* u modelu *AccessLog* prilikom spremanja u bazu podataka, radni okvir zna kako pretvoriti objekte i vrijednosti u tipove podataka koje baza podataka razumije.

3.2.3. Pregled dnevnika pristupa

Korištenjem sefa puni se pristupni dnevnik. No, koliko god zapisa postojalo, od njih nema pretjerane koristi ako se do njih ne može doći. Zato valja izložiti te podatke grafičkom sučelju.

Potrebno je vratiti sve zapise dnevnika pristupa u JSON formatu sortirane silazno po datumu stvaranja (prvo najnoviji zapisi). Zapisi su dostupni na putanji /**api**/**access-log** kroz **GET** metodu,

```
Route::get('/access-log', FetchAccessLogAction::class);
```

a implementirana je u **FetchAccessLogAction** metodi:

```
class FetchAccessLogAction
{
    public function __invoke(Request $request): JsonResponse
    {
        $accessLogEntries = AccessLog::query()->orderByDesc('at_time')->get();

        return response()->json(['data' => $accessLogEntries]);
    }
}
```

Važno je naglasiti da se zapisi ne povezuju s entitetom "Kartica" već je to delegirano na grafičko sučelje. Kako korisnik može izmjenjivati i brisati kartice, nakon bilo koje akcije bilo bi potrebno osvježiti podatke iz dnevnika pristupa što generira nepotreban mrežni promet. S obzirom da je logika povezivanja ovih entiteta vrlo jednostavna (oba entiteta moraju sadržavati isti *UID*) sigurno je implementirati funkcionalnost u grafičkom sučelju.

3.2.4. Autorizacija korisnika

Sadržaj dnevnika pristupa i upravljanje karticama nije javno dostupno. Korisnik je dužan autorizirati se pomoću svojih pristupnih podataka (korisničko ime i lozinka) prije poduzimanja bilo kojih drugih radnji.

Autorizacija korisnika je dostupna na putanji /**api**/**login** kroz **POST** metodu.

```
Route::post('login', LoginAction::class);
```

Prije implementiranja autorizacije potrebno je pohraniti korisničke pristupne podatke. Za to će poslužiti jednostavan entitet "Korisnik" koji sadrži podatke o korisničkom imenu i lozinki. Poput prethodnih entiteta i ovaj entitet je vrlo jednostavan pa nema potrebe za ER dijagramom, samo migracijski kod:

```
class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('username')->unique();
            $table->string('password');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Potreban je i model **User**, no ovaj put se ne nasljeđuje klasa *Model* već klasa *Authenticatable*. Ta klasa sadrži određene metode potrebne za autoriziranje korisnika pa radni okvir pomoću nje zna kako autorizirati korisnika i stvoriti sesiju.

```
class User extends Authenticatable
{
}
```

Radni okvir sadrži korisnu metodu **Auth::attempt** koja prima jedan parametar tipa polje. U polje se postave vrijednosti korisničkog imena i lozinke koje je poslao korisnik i ako su podaci točni, korisnik je autoriziran. Ovisno o ishodu metode vraća se prikladna poruka u JSON formatu.

```
class LoginAction extends Controller
{
    public function __invoke(Request $request): JsonResponse
    {
        $username = $request->username;
        $password = $request->password;

        if (Auth::attempt(['username' => $username, 'password' => $password]) ===
            false) {
            return response()->json(['message' => 'Korisnicko ime i/ili lozinka su
                ne ispravni.'], 401);
        }

        return response()->json(['message' => 'Uspjesna prijava']);
    }
}
```

3.3. Grafičko sučelje

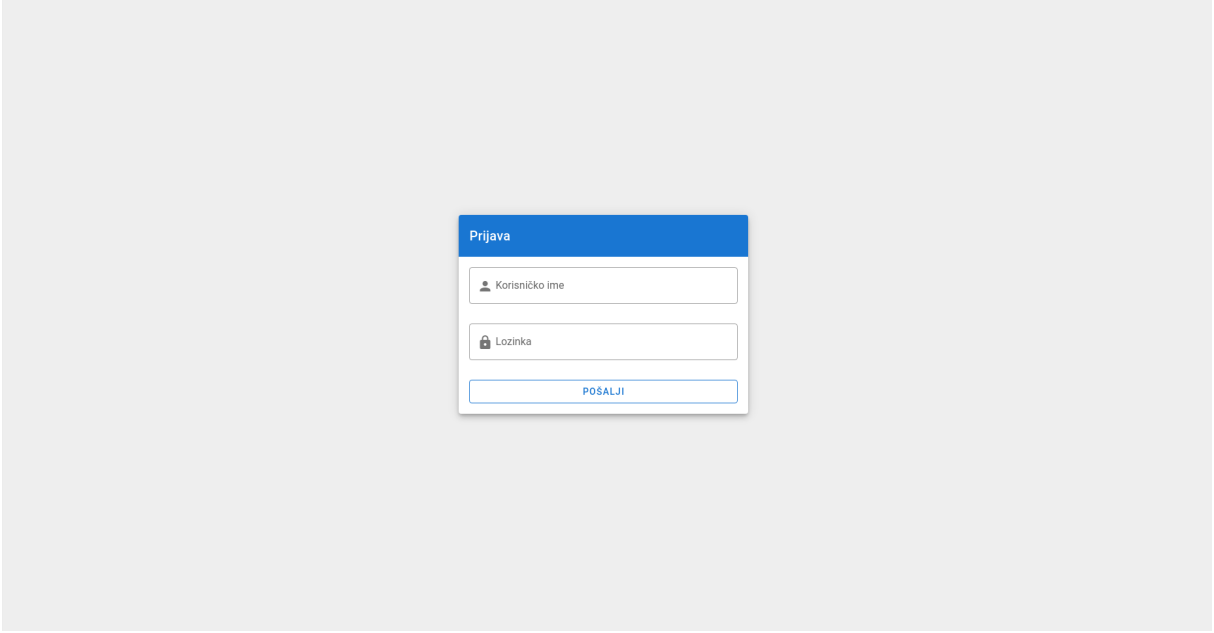
Grafičko sučelje predstavlja prikladan način za korisnika kako bi jednostavno obavljao zadatke upravljanja karticama i pregleda dnevnika pristupa. Prethodno definirane funkcionalnosti u pozadinskoj aplikaciji koje treba implementirati u grafičkom sučelju su:

- Autorizacija korisnika
- Upravljanje karticama
- Pregled dnevnika pristupa

Aplikacije na web platformi koje se izvršavaju u pregledniku nemaju mogućnost odabira jezika. Tri su jezika potrebna za izradu grafičkog sučelja na web platformi: HTML, CSS i JavaScript. Pri razvoju sučelja koristi se radni okvir *Vue*, a osnovne vizualne komponente pruža *Vuetify*.

3.3.1. Autorizacija korisnika

Prije poduzimanja bilo kojih akcija, korisnik se mora autorizirati (prijaviti) u aplikaciju. Forma za unos pristupnih podataka prikazana je u nastavku.



Slika 3: Forma za unos pristupnih podataka

Na slici 3 su tri važne komponente za obavljanje autorizacije:

1. Polje za unos korisničkog imena

```
<v-text-field
    v-model="username.value"
    label="Korisnicko ime"
    prepend-inner-icon="mdi-account"
    outlined
/>
```

2. Polje za unos lozinke

```
<v-text-field
    v-model="password.value"
    type="password"
    label="Lozinka"
    prepend-inner-icon="mdi-lock"
    outlined
/>
```

3. Gumb za slanje zahtjeva

```
<v-btn
    color="primary"
    type="submit"
    block
    outlined
>
    Posalji
</v-btn>
```

Komponente su omotane oko *v-form* komponente koja je zaslužna za slanje zahtjeva kad korisnik pritisne gumb.

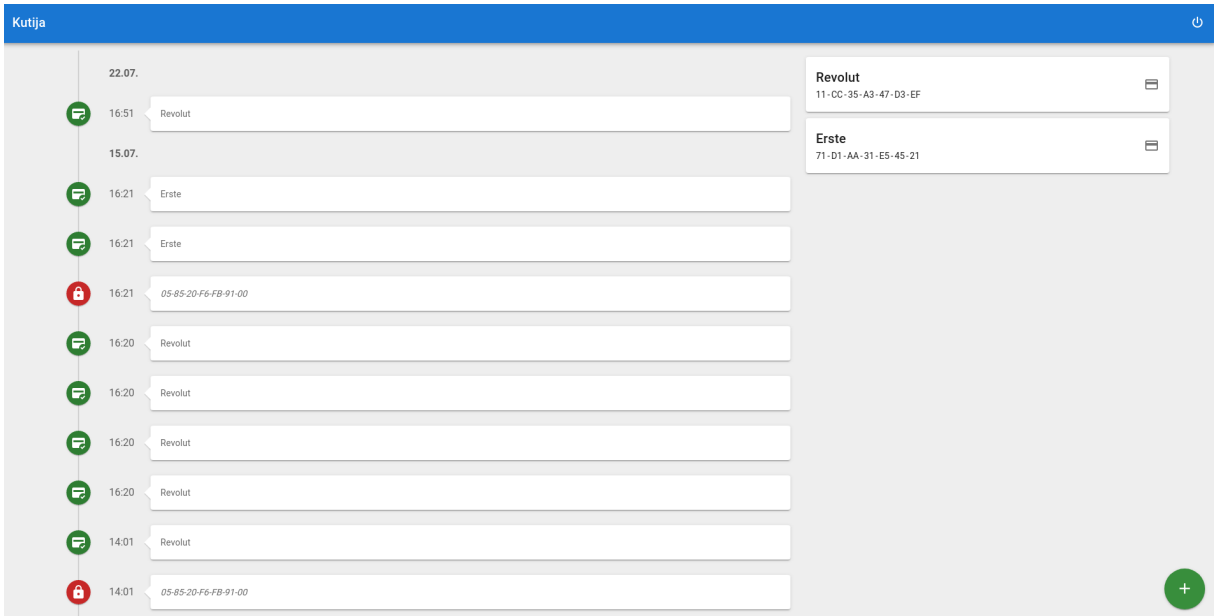
```
<v-form @submit.prevent="login">
</v-form>
```

Kako HTML ne može slati zahtjeve prema serveru i izvoditi bilo kakvu logiku, funkcija *login* koja se okida na *submit* događaj je implementirana pomoću JavaScript jezika.

```
import axios from 'axios'

export default {
  name: 'LoginForm',

  data: () => ({
    username: {
      value: ''
    },
    password: {
      value: ''
    }
```

```
  }),

  methods: {
    async login () {
      const { data } = await axios.post('/api/login', { username: this.username.
          value, password: this.password.value })
      const serverResponse = data.data

      this.loginSuccessful(serverResponse)
    },

    loginSuccessful (messageForUser) {
      this.$emit('success', messageForUser)
    }
  }
}
```

Svaka komponenta ima naziv **name**, može sadržavati podatke **data**, implementirati metode **methods** i sl. U funkciji *data* se nalazi objekt s vrijednostima polja koja korisnik mora ispuniti. U *methods* objektu je jedna metoda **login** koja je zadužena za slanje zahtjeva na pozadinsku aplikaciju. Koristi se biblioteka **axios** jer olakšava slanje zahtjeva naspram standardnih *fetch* i *XMLHttpRequest* funkcija. Nakon uspješne autorizacije metoda *$emit* emitira događaj i obavještava nadređenu komponentu. Tada nadređena komponenta zna da može učitati pristupni dnevnik i kartice.



Slika 4: Pristupni dnevnik i kartice

3.3.2. Upravljanje karticama

Za prikaz svih kartica potrebne su dvije komponente: komponenta za dohvaćanje kartica sa servera i komponenta za prikaz istih.

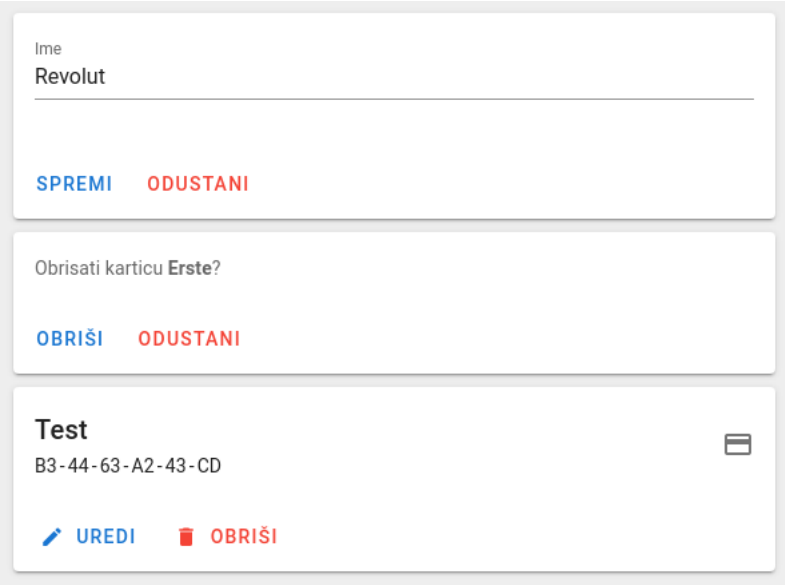
```
<CardListData>
  <template v-slot:default="{ cards, addCard, removeCard, updateCard }">
    <CardList
      :cards="cards"
      @card-created="addCard"
      @card-deleted="removeCard"
      @card-updated="updateCard"
    />
  </template>
</CardListData>
```

Komponenta **CardListData** se brine o načinu dohvaćanja podataka, sprema ih u varijablu *cards* kao polje s podacima kartica.

Kako je kartice moguće uređivati, brisati i dodavati nije prikladno da komponenta koja konzumira varijablu *cards* ujedno i izmjenjuje istu. Kako bi pravilno napravila izmjene, komponenta **CardList** mora znati unutarnje ponašanje komponente **CardListData**. To dovodi do curenja apstrakcije (eng. *abstraction leak*) pa iz tog razloga postoje tri metode: *addCard*, *removeCard*, *updateCard*. Na taj način komponenta *CardListData* pruža stabilan API. Komponenta *CardList* mora znati potrebne argumente za svaku funkciju, a ne mora znati na koji način će se promjene izvesti.

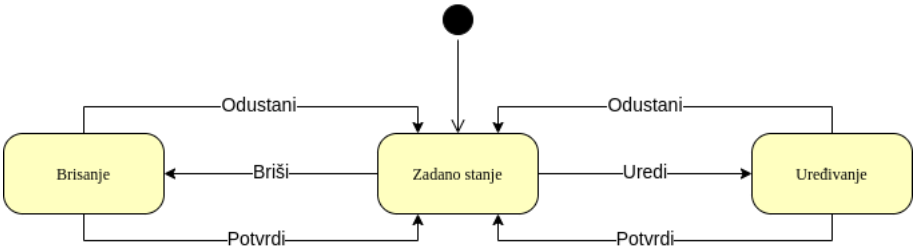
Komponenta **CardList** pomoću petlje jednostavno prikazuje sve kartice. Kako je svaku karticu moguće mijenjati i brisati potrebna je dodatna komponenta: **CardListItem**. Ona zahtjeva naziv i UID kartice, a ima tri stanja u kojima se može nalaziti:

1. Zadano stanje - samo prikazuje naziv i UID te gumbe za uređivanje ili brisanje
2. Stanje uređivanja - polje za unos naziva i gumb za spremanje ili odustajanje
3. Stanje brisanja - gumb za potvrdu brisanja i gumb za odustajanje



Slika 5: Tri stanja kartice

Na slici 5 prva kartica je u stanju uređivanja, druga je u stanju brisanja, a posljednja je u zadanom stanju. Iako u programskom kodu nije eksplicitno definirana, postoji implicitna mašina stanja (eng. *state machine*). Mašina stanja je koncept ponašanja programa definiran kroz konačan broj stanja i tranzicija. Svako stanje pruža relevantne informacije o programu, dok tranzicije ukazuju na moguće prijelaze između stanja [4]. Jedna od prednosti mašine stanja je mogućnost kreiranja dijagrama stanja i prijelaza.



Slika 6: Tri stanja kartice

Na slici 6 prikazan je dijagram mašine stanja jedne kartice. Zadano stanje je ujedno i početno stanje (eng. *initial state*) i jedino iz tog stanja se može prijeći u druga dva uz pomoć tranzicija "Briši" i "Uredi". Druga dva stanja imaju po dvije tranzicije "Potvrdi" i "Odustani" koje vode u zadano stanje. Prema dijagramu nije moguće direktno prijeći iz stanja brisanja u stanje uređivanja i obratno.

Svako stanje zahtjeva nešto drugačije sučelje i elemente pa je pogodno kreirati zasebne komponente za svako stanje. Tri su komponente potrebne:

1. **CardListItemDefaultState**
2. **CardListItemEditState**
3. **CardListItemDeleteState**

Sve tri komponente su omotane oko komponente **CardListItem**. Ona se brine o trenutnom stanju kao i o prelasku u druga stanja pa je to čini "mašinom" u mašini stanja.

```
<CardItemEditState
    v-if="inEditMode"

    :id="id"
    :name="name"

    @cancel="stopEdit"
    @success="completeEdit"
/>
<CardItemDeleteState
    v-else-if="inDeleteMode"

    :id="id"
    :name="name"

    @cancel="stopDelete"
    @success="completeDelete"
/>
<CardItemDefaultState
    v-else

    :name="name"
    :uid="uid"

    @edit-requested="startEdit"
    @delete-requested="startDelete"
/>
```

Direktive *v-if*, *v-else-if*, *v-else* se brinu da u jednom trenutku korisniku bude vidljiva samo jedna komponenta (jedno stanje). Na događaje (*@success*, *@cancel*, ...) izvršavaju se metode (*startEdit*, *stopDelete*, ...) i one predstavljaju tranzicije - prelazak iz jednog stanja u drugi.

```
data: () => ({
    editActive: false
}),

computed: {
    inEditMode () {
      return this.editActive === true
    },
},

methods: {
    startEdit () {
      this.editActive = true
    },

    stopEdit () {
      this.editActive = false
    }
}
```

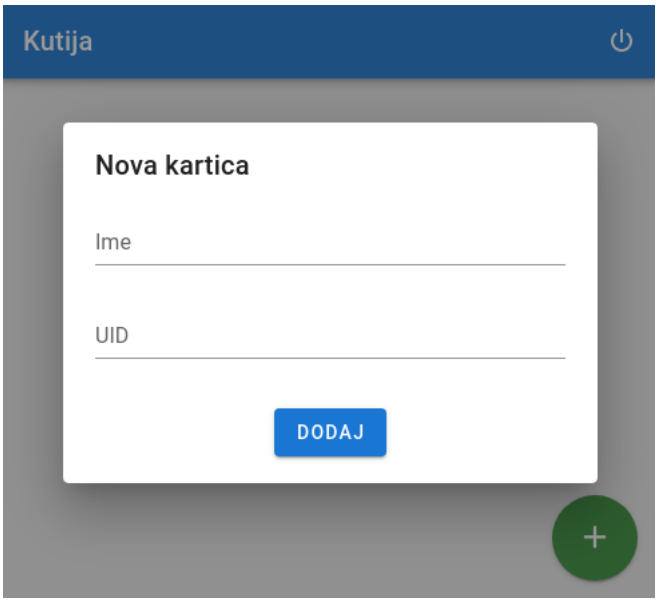
Sve tri komponente su trivijalne - uglavnom definiraju izgled i nemaju previše logike. Važno je naglasiti da u komponenti *CardListItemEditState* kad korisnik klikne na gumb "Spremi" šalje se zahtjev na pozadinsku aplikaciju koji ažurira naziv kartice.

```
axios.put('/api/card/' + this.id, { name: this.newName.value })
```

Slična je situacija i s *CardListItemDeleteState* komponentom. Kada korisnik klikne na gumb "Obriši" šalje se zahtjev na pozadinsku aplikaciju koji briše karticu.

```
axios.delete('/api/card/' + this.id)
```

Preostalo je implementirati dodavanje novih kartica.



Slika 7: Dodavanje nove kartice

Na slici 7 prikazan je prozor s formom za unos nove kartice. Poput forme za autorizaciju korisnika, dodavanje nove kartice ima tri važne komponente:

1. Polje za unos naziva kartice

```
<v-text-field
    v-model="name.value"

    label="Ime"
    hint="npr. Glavna kartica"
    maxlength="30"
/>
```

2. Polje za unos jedinstvenog identifikatora (UID)

```
<v-text-field
    v-model="uid.value"

    label="UID"
    hint="npr. A5-C1-23-13-1C-7E"
    maxlength="40"
/>
```

3. Gumb za slanje zahtjeva

```
<v-btn
    color="primary"
    type="submit"
>
    Dodaj
</v-btn>
```

Komponenta za slanje zahtjeva prilikom pritiska gumba:

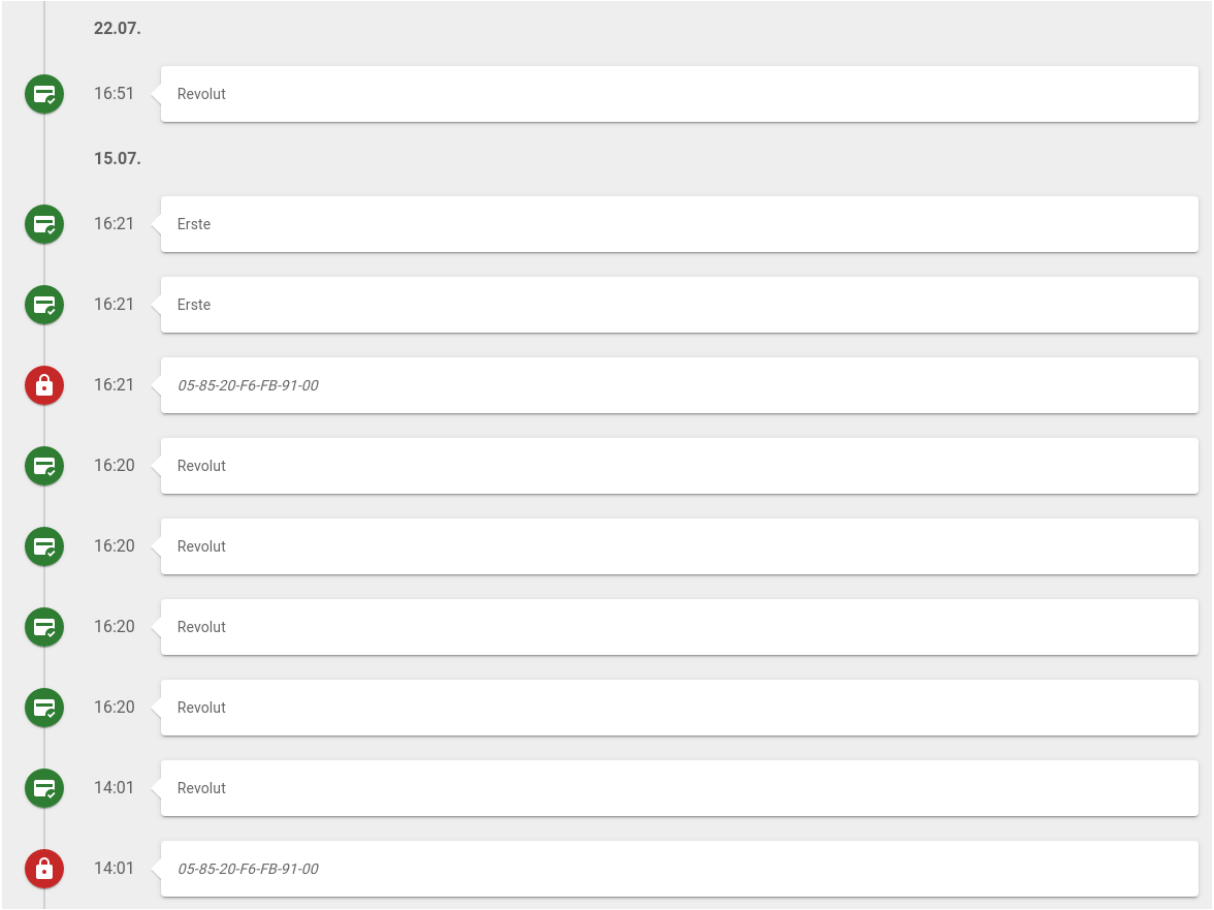
```
<v-form @submit.prevent="submitNewCard">
</v-form>
```

Ove komponente i skripta za slanje zahtjeva čine komponentu **CardListNewItem**:

```
export default {
  name: 'CardListNewItemForm',

  data: () => ({
    name: {
      value: ''
    },
    uid: {
      value: ''
    }
  }),

  methods: {
    async submitNewCard () {
        const { data } = await axios.post('/api/card', {
          name: this.name.value,
          uid: this.uid.value
        })

        const { id, name, uid } = data.data
        this.cardCreated({ id, name, uid })
    }
  }
}
```

3.3.3. Pregled dnevnika pristupa

Dnevnik pristupa implementiran je kao vremenska crta dozvoljenih i odbijenih pristupa. Svaka stavka ima vrijeme pristupa, naziv ili UID kartice te status (dozvoljeno ili odbijeno). Stavke su grupirane po datumu.



Slika 8: Vremenska crta dnevnika pristupa

Potrebne komponente su podijeljenje u dvije grupe:

1. Komponente za obradu podataka
2. Komponente za prikaz podataka

Komponenta **AccessLogData** brine se o preuzimanju podataka sa servera. Preuzete podatke daje na korištenje ostalim komponentama kroz polje *entries*.

```
export default {
  name: 'AccessLogData',

  data: () => ({
    entries: []
  }),

  created () {
    this.fetchAccessLogEntries()
  },

  methods: {
    async fetchAccessLogEntries () {
      const { data: { data } } = await axios.get('/api/access-log')
```

```
      this.entries = data
    }
  }
}
```

Prethodno je rečeno da kartice nisu povezane s dnevnikom pristupa u pozadinskoj aplikaciji već kroz grafičko sučelje. Kako se u vremenskoj crti prikazuje naziv kartice ako ona postoji (u suprotnom se prikazuje UID), brisanje ili dodavanje kartica zahtjeva promjene vremenske crte. Također, potrebno je grupirati stavke po datumu. Ovi zahtjevi su implementirani u komponenti **AccessLogBridge**.

```
const yearFormat = 'yyyy'
const dateFormat = 'dd.MM.' + yearFormat

computed: {
  timelineGroups () {
    const entriesGroupedByDate = groupBy(this.entries, entry => format(entry.
        timestamp(), dateFormat))

    const groupedEntriesWithSimplifiedDates = mapKeys(entriesGroupedByDate, (_, key)
         => this.simplifyDate(key))

    return mapValues(groupedEntriesWithSimplifiedDates, entries => entries.map(this.
        convertEntryToTimelineItem))
  }
},

methods: {
  createTimelineItemContent (entry) {
    const entryUid = entry.uid()

    const findCardByEntryUid = () => this.cards.find(card => card.uid === entryUid)
        || null

    const cardName = findCardByEntryUid()?.name || null

    return cardName ?? entryUid
  },

  convertEntryToTimelineItem (entry) {
    return new TimelineListItemModel(
      entry.id,
      format(entry.timestamp(), 'HH:mm'),
      this.createTimelineItemContent(entry),
      entry instanceof AccessGrantedEntry
    )
  },

  simplifyDate (date) {
    const today = format(new Date(), dateFormat)
    const yesterday = format(startOfYesterday(), dateFormat)
    const currentYear = format(new Date(), yearFormat)
```

```
    if (date === today) return 'Danas'
    else if (date === yesterday) return 'čJuer'
    else if (date.substr(-yearFormat.length) === currentYear) return date.substring
        (0, date.length - yearFormat.length)

    return date
  }
}
```

Obrađeni podaci prosljeđuju se komponenti za prikaz vremenske crte. To je komponenta **AccessLogTimeline** koja uz pomoć petlje prikazuje sve grupe.

```
<TimelineGroup
    v-for="(items, date) of groups"
    :key="date"

    :title="date"
    :items="items"
/>
```

Svaka grupa ima više stavki pa je prikaz istih delegiran na **TimelineGroup** komponentu.

```
<v-timeline-item hide-dot>
    <span class="text-uppercase text--secondary font-weight-bold is-size-5">{{ date
        }}</span>
</v-timeline-item>

<TimelineList :items="items" />
```

Prvi element je datum, ponaša se kao naslov grupe. Komponenta **TimelineList** zadužena je za prikaz svih stavki jedne grupe.

```
<TimelineItem
    v-for="item in internalItems"
    :key="item.id"

    :success="item.success"
    :time="item.time"
>
    <span :class="{ 'font-italic': !item.success }">{{ item.content }}</span>
</TimelineItem>
```

Pomoću petlje prikazuje više **TimelineItem** komponenti. Ta komponenta prikazuje vrijeme, sadržaj i dvije različite ikone i boje ovisno o (ne)uspješnosti pristupa.

```
<v-timeline-item
  :icon="success ? 'mdi-credit-card-check' : 'mdi-lock'"
  fill-dot
  :color="(success ? 'green' : 'red') + ' darken-3'"
>
  <div class="text--secondary">{ time }</div>
  <v-card class="custom-timeline-item-card">
    <v-card-text>
          <span>
```

```
            <slot />
          </span>
    </v-card-text>
  </v-card>
</v-timeline-item>
```

Rezultat je prikaz dnevnika pristupa odnosno povijest (ne)uspješnih pristupa (slika 8).

Aplikacija za kontrolu pristupa je dovršena, a sljedeći korak je sef koji će aplikaciji dati smisao i način punjenja dnevnika pristupa.

4. Sef

Sef predstavlja posljednju komponentu sustava za kontrolu pristupa. Sastoji se od nekolicine elektroničkih komponenti i upravljačkog softvera (eng. *firmware*).

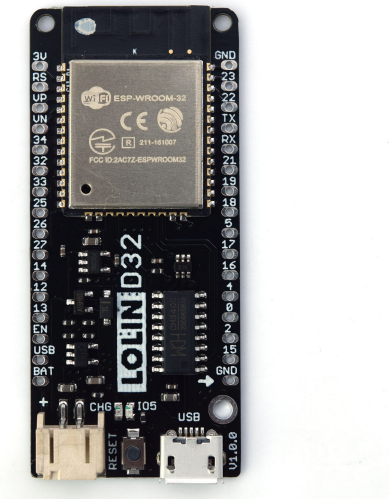
U ovom poglavlju komponente i softver će biti opisani i razrađeni.

4.1. Komponente

Sef zahtjeva različite komponente kako bi ispunio svoju zadaću. Komponente su povezane s mikrokontrolerom koji upravlja s istima. U nastavku je popis potrebnih komponenti s kratkim opisom.

4.1.1. ESP32

Koristi se izvedba ESP32 pločice tvrtke *WEMOS*, model *D32*. Sadrži 22 digitalna i 8 analognih pinova, a radi na 3.3V. Procesorska jedinica radi na brzini do 240MHz.

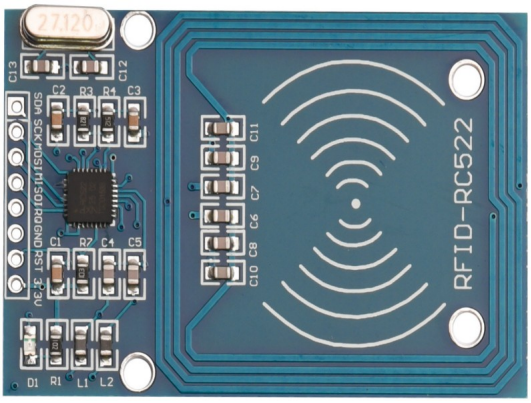


Slika 9: ESP32 pločica (Izvor: [5])

Uz glavnu pločicu potreban je i takozvani *power shield* koji omogućava napajanje od 12V. Komponenta nije uključena u standardni paket WEMOS D32, a potrebna je zbog električne brave koja radi na 12V.

4.1.2. MFRC522

Kreditne kartice sadrže RFID čip u kojem je pohranjen jedinstveni identifikator kartice (UID). Za autorizaciju pristupa sefu potrebno je pročitati UID s čitačem RFID kartica.



Slika 10: MFRC522 čitač (Izvor: [6])

4.1.3. Mikroprekidač

Magnetski mikroprekidač je dvodijelna komponenta koja zatvara strujni krug kad su obje komponente blizu jedna drugoj (oko 20 milimetara). Jednostavna komponenta koja služi pri određivanju stanja sefa (otvoren ili zatvoren). Proizvođač je tvrtka *SparkFun*.

Slika 11: Magnetski mikroprekidač (Izvor: [7])

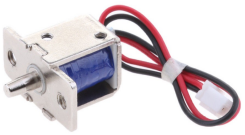
4.1.4. RGB LED

LED dioda koja podržava RGB spektar boja. Komponenta služi kao informacija korisniku o uspješnosti otvaranja sefa.

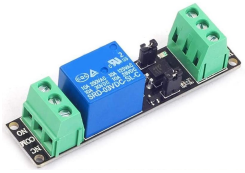
Slika 12: RGB LED (Izvor: [8])

4.1.5. Brava i relej

Električna brava koja radi na 12V i 500mA. Kad relej zatvori strujni krug brava se otvara i omogućava korisniku pristup sefu.



(a) Brava (Izvor: [9])



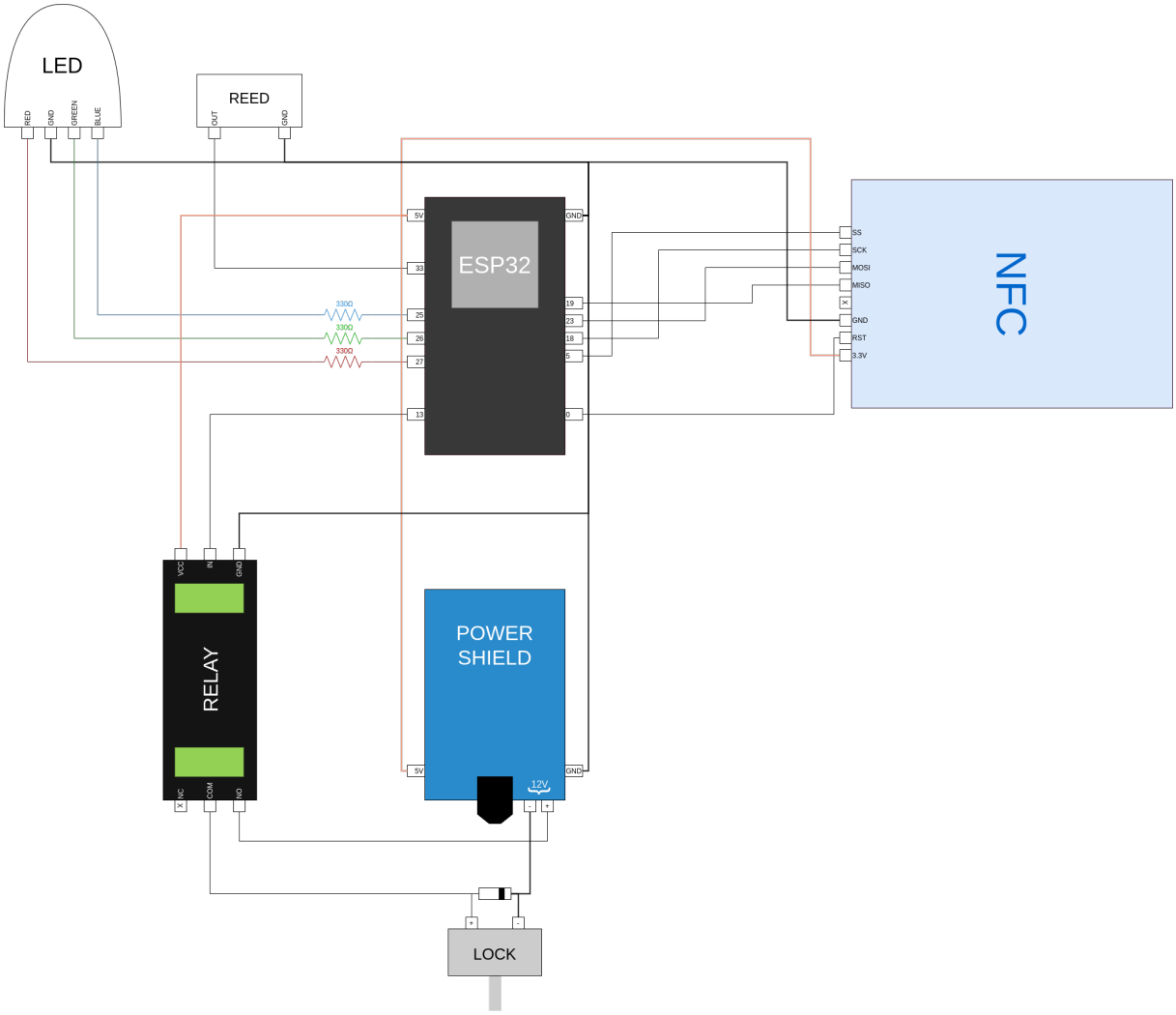
(b) Relej (Izvor: [10])

4.1.6. Napajanje

Većina komponenti zahtjeva 3.3V napajanje, no električna brava radi na 12V pa zahtjeva vanjsko napajanje. Potrebno je vanjsko napajanje koje pretvara 220V u 12V.

Slika 14: Napajanje (Izvor: [11])

4.1.7. Schema komponenti



Slika 15: Schema komponenti

Na slici 15 u centru svega nalazi se ESP32 mikrokontroler. Pinovi s desne strane mikrokontrolera (19, 23, 18, 5 i 0) potrebni su za komunikaciju s NFC čitačem. Sve tri boje LED diode (crvena, zelena i plava) su u upotrebi, a između svake žice i mikrokontrolera nalazi se otpornik od 300 oma. Magnetski mikroprekidač najjednostavnija je komponenta, a informacije o stanju pruža mikrokontroleru na pinu broj 33. Upravljanje bravom se ne izvodi direktno već pomoću releja na pinu broj 13. Relej i brava zahtijevaju napajanje i uzemljenje od 12V s *power shield-a*. Dovodom struje na *IN* ulaz releja zatvara se strujni krug i propušta se struja na *COM* izlaz koji je povezan s pozitivnim polom brave i otključava bravu.

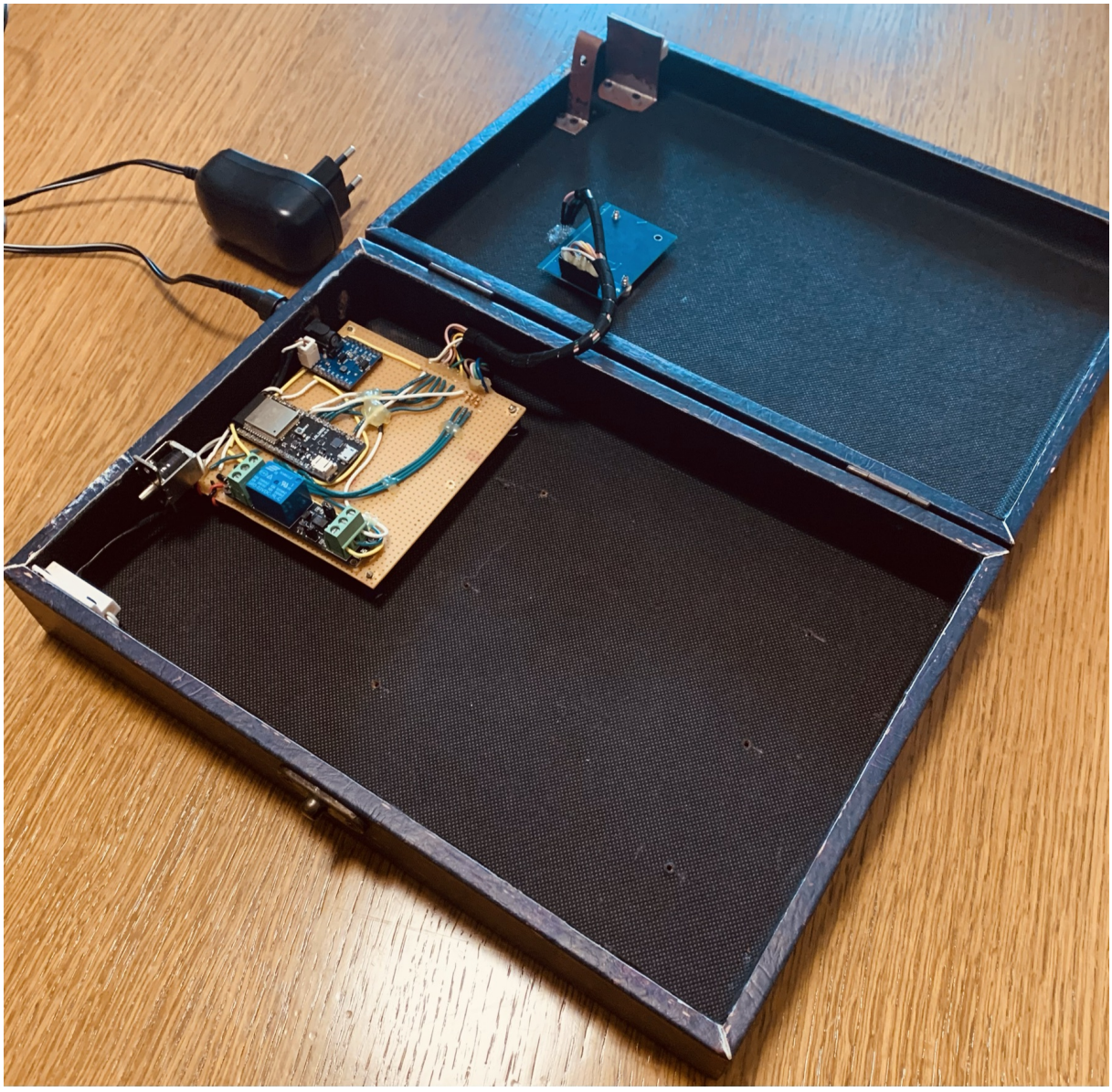
Testiranjem sefa utvrđeno je neočekivano ponašanje. Uzastopno otključavanje brave dovodi do nemogućnosti čitanja kartice i podataka s iste. Ispitivanjem je utvrđeno da mikrokontroler radi pravilno, dok NFC čitač ne prepoznaje ranije prepoznatu karticu. Nema uzorka u pojavljivanju ovakvog ponašanja (npr. nakon X uspješnih čitanja ili X minuta dolazi do nepravilnog rada) već se događa nasumično.

Brava uvlači klin pomoću elektromagnetske induktivne zavojnice. Dovodom struje na

zavojnicu električna energija se pretvara u magnetsku i tvori magnetsko polje. Odvodom struje s polova brave magnetsko polje se ruši i stvara naponski impuls obrnutog polariteta [12]. Elektroni promjene smjer kretanja i, na sreću, samo privremeno poremete rad NFC čitača. Nemoguće je spriječiti promjenu smjera kretanja, no moguće je ograničiti kretanje. Za to je potrebna takozvana *flyback* dioda koja propušta jedan smjer, a blokira obrnuti smjer kretanja [12]. Na slici 15 kod kontakata brave nalazi se *flyback* dioda. Na taj način elektroni s obratnim smjerom kretanja ne dolaze do ostalih komponenti sefa i NFC čitač radi pravilno.

4.1.8. Integracija komponenti u sef

Mikrokontroler, *power shield*, relej, *flyback* dioda i ožičenje zalemljeni su na plastičnu pločicu, a pločica je pričvršćena za kutiju. NFC čitač, brava i utor za bravu, mikroprekidač i RGB LED dioda direktno su pričvršćeni za kutiju.



Slika 16: Komponente integrirane u kutiju

4.2. Upravljački softver

Upravljački softver (eng. *firmware*) kontrolira komponente i komunicira s aplikacijom prilikom autorizacije kartice. Napisan je u *C++* programskom jeziku i *Arduino* radnom okviru. Kompiliranje izvornog koda i učitavanje binarnog koda u mikrokontroler omogućava *PlatformIO*.

4.2.1. Konfiguracija pinova i bežične mreže

Prema schemi komponenti (slika 15) definirane su vrijednosti pinova:

```
#define RST_PIN     0
#define SS_PIN      5

#define LOCK_PIN    13
#define STATE_PIN   33

#define RED_PIN     27
#define GREEN_PIN   26
#define BLUE_PIN    25

#define ACCESS_POINT_NAME       "BoxAP"
#define ACCESS_POINT_PASSWORD   "********"
```

Potrebno je instancirati određene objekte:

```
BoxAuthorizer authorizer = BoxAuthorizer();
Box box = Box(LOCK_PIN, STATE_PIN, authorizer);
CardReader reader = CardReader(SS_PIN, RST_PIN);
StatusLED LED = StatusLED(GREEN_PIN, RED_PIN, BLUE_PIN);
```

Mikrokontroler zahtjeva dvije funkcije u glavnom programu: **setup** i **loop**. Funkcija *setup* izvodi se samo jednom, pri paljenju mikrokontrolera. Funkcija *loop* izvršava se više puta (poput *for* ili *while* petlje) sve dok se mikrokontroler ne ugasi.

Funkcija *setup* je pogodna za inicijalnu pripremu i konfiguraciju pinova i povezivanje na bežičnu mrežu.

```
void setup() {
    Serial.begin(9600);

    box.configurePins();

    LED.configurePins();
    LED.idle();

    reader.begin();
    reader.onSuccessfulAttempt(tryToAuthorizeAccess)
          .onFailedAttempt(indicateCardReadingFailure)
          .onAnyAttempt(resetLED);

    NetworkManager manager = NetworkManager(ACCESS_POINT_NAME, ACCESS_POINT_PASSWORD
        );
```

```
    manager.connect([]() {
        Serial.println("Successfully connected to the network!");
    });
}
```

Registriraju se i tri *callback-a* nad čitačem kartica:

1. Pokušaj čitanja kartice uspješan
2. Pokušaj čitanja kartice neuspješan
3. Callback koji se uvijek izvršava nakon prva dva

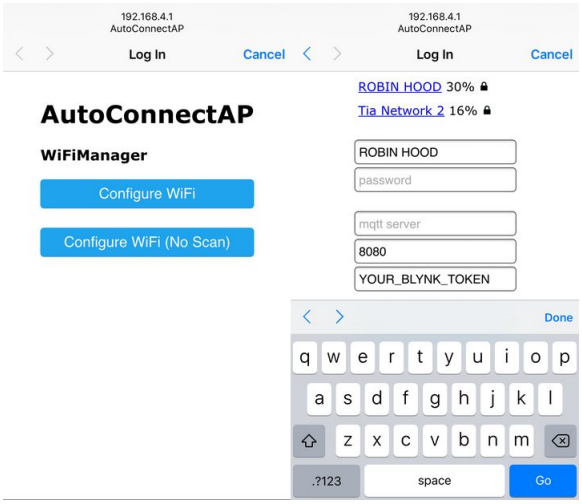
Metode *configurePins* objekata *box* i *LED* konfiguriraju način rada pinova: ulazni ili izlazni. Kako se bojama RGD LED diode može prilagoditi intenzitet svjetlosti konfiguracija pinova je kompleksnija.

```
pinMode(_lockPin, OUTPUT);
pinMode(_statePin, INPUT_PULLUP);

ledcSetup(_greenChannel, 5000, 8);
ledcAttachPin(_greenPin, _greenChannel);

ledcSetup(_redChannel, 5000, 8);
ledcAttachPin(_redPin, _redChannel);

ledcSetup(_blueChannel, 5000, 8);
ledcAttachPin(_bluePin, _blueChannel);
```

Klasa *NetworkManager* implementira jednu metodu *connect*. Pomoću biblioteke *WiFiManager* počinje proces povezivanja na bežičnu mrežu.

```
void NetworkManager::connect(void (*onSuccess)()) const {
    WiFiManager manager;
    bool connectionSuccessful = manager.autoConnect(accessPoint.name, accessPoint.
        password);

    if (connectionSuccessful) {
        onSuccess();
    }
}
```

Ukoliko je dostupna mreža na koju se mikrokontroler povezao u prošlosti automatski se povezuje na nju. U suprotnom mikrokontroler se ponaša kao pristupna točka (eng. *access point*) i zahtjeva manualan odabir bežične mreže i lozinke. Pomoću drugog uređaja (npr. *smartphone*) potrebno je povezati se na mrežu imena *ACCESS_POINT_NAME* s lozinkom *ACCESS_POINT_PASSWORD*, a potom se otvara portal kroz koji se konfigurira mreža na koju će se mikrokontroler povezivati ubuduće.



Slika 17: Portal za odabir mreže (Izvor: [13])

4.2.2. Čitanje kartice

Nakon uspješne konfiguracije bežične veze funkcija *setup* završava te započinje konstantno izvršavanje funkcije *loop*.

```
void loop() {
    reader.tryReadingTheCard();
    delay(1000);
}
```

Metoda *tryReadingTheCard* provjerava je li kartica prislonjena blizu čitača i pokušava pročitati UID. Ovisno o uspješnosti čitanja UID-a izvršava *callback-ove* definirane u *setup* funkciji. Biblioteka *MFRC522* olakšava rad s NFC čitačem i instancirana je u konstruktoru klase.

```
CardReader::CardReader(byte chipSelectPin, byte resetPowerDownPin) {
    reader = MFRC522(chipSelectPin, resetPowerDownPin);
}

void CardReader::tryReadingTheCard() {
    if (reader.PICC_IsNewCardPresent() == false)
        return;

    if (reader.PICC_ReadCardSerial() == false) {
        failedAttemptCallback();
        anyAttemptCallback();
        return;
    }

    Card card = Card(reader.uid);

    reader.PICC_HaltA();
    reader.PCD_StopCrypto1();

    if (card.isUidValid()) {
        successfulAttemptCallback(card);
    }
```

```
    else {
        failedAttemptCallback();
    }

    anyAttemptCallback();
}
```

Instancira se objekt *Card* koji se prosljeđuje *callback-u* uspješnog čitanja. Konstruktor prima jedan argument tipa *MFRC522::Uid* i pretvara ga u *String*. Klasa pruža metode za provjeru valjanosti UID-a (*isUidValid*) kao i pretvaranje cijelog objekta *Card* u *UID String* (*toUid*).

```
Card::Card(MFRC522::Uid uid) {
    UID = uidToHexString(uid);
}

String Card::uidToHexString(MFRC522::Uid uid) {
    if (uid.size == 0) return "";

    String hexString = "";

    for(unsigned short int i = 0; i < uid.size; i++) {
        const char prefix = uid.uidByte[i] < 10 ? '0' : '\0';

        String byteAsHexString = prefix + String(uid.uidByte[i], HEX);
        hexString += byteAsHexString + " ";
    }

    hexString.trim();
    hexString.toUpperCase();
    hexString.replace(' ', '-');

    return hexString;
}

bool Card::isUidValid() {
    return UID.isEmpty() == false;
}

String Card::toUid() const {
    return UID;
}
```

4.2.3. Autorizacija pristupa

Uspješnim čitanjem kartice izvršavanje programa se nastavlja u funkciji *tryToAuthorizeAccess*.

```
void tryToAuthorizeAccess(const Card& card) {
    Serial.println("-------------------------------------------------");
    Serial.println("Detected new card with UID: " + card.toUid());

    LED.flashGreen(1);
```

```
    delay(1000);
    LED.idle();

    if (box.isOpened()) {
        Serial.println("Box is opened, skipping...");
        LED.flashRed(1);
        return;
    }

    Serial.println("Box is closed, authorizing card with the server...");

    box.authorize(card)
       .onSuccess(grantAccess)
       .onFailure(notifyForbiddenAccess);
}
```

Jednim zelenim bljeskom LED diode korisniku se ukazuje na uspješno čitanje kartice. Sljedeći je korak utvrditi je li kutija već otvorena (pomoću magnetskog mikroprekidača), ako je potrebno je jednim crvenim bljeskom informirati korisnika, a izvršavanje obustaviti.

```
bool Box::isOpened() const {
    return digitalRead(_statePin);
}
```

Zatvorena kutija je uvjet za slanje autorizacijskog zahtjeva na udaljeni server.

Slanje zahtjeva je izvedeno u pomoćnoj klasi *POST* koja apstrahira rad s bibliotekama *HTTPClient* i *WiFiClient*. Klasa ne zna unaprijed gdje se server nalazi, koji podaci će se poslati i sl. već zna **kako** poslati zahtjev.

```
POST::POST(): _wifiClient(), _httpClient() {
}

POST::~POST() {
    _httpClient.end();
}

POST POST::request() {
    return {};
}

POST& POST::to(const String& url) {
    _httpClient.begin(_wifiClient, url);
    _httpClient.addHeader("Content-Type", "application/x-www-form-urlencoded");

    return *this;
}

POST& POST::withPayload(const String& payload) {
    _payload = payload;

    return *this;
}
```

```
uint8_t POST::responseCode() {
    return _httpClient.POST(_payload);
}
```

Metode *POST* klase su iskorištene u metodi *BoxAuthorizer::authorize*. Ova metoda ne zna kako poslati zahtjev ali zna **gdje** i **koje** podatke treba poslati. Prosljeđuje se URL adresa udaljenog servera i UID. Ukoliko je HTTP kod odgovara jednak 200 autorizacija je uspješna.

```
BoxAuthorizer::Result BoxAuthorizer::authorize(const String &uid) {
    return POST::request()
                .to("http://144.126.244.106/api/box/authorize")
                .withPayload("uid=" + uid)
                .responseCode() == 200
                                ? authorizationSucceeded()
                                : authorizationFailed();
}
```

4.2.4. Otključavanje sefa

Uspješnom autorizacijom izvršavanje programa se nastavlja u funkciji *grantAccess*. Dupli bljesak zelene boje je indikator korisniku da je autorizacija uspješna i sef se otključava na četiri sekunde.

```
void grantAccess() {
    Serial.println("Successful authorization! Opening the box...");

    LED.flashGreen(2);

    box.unlock();
    delay(4000);
    box.lock();
}
```

Metode za otključavanje i zaključavanje sefa definirane su u *Box* klasi na način da pin koji je spojen na relej postave na **HIGH** ili **LOW**. Mikrokontroler propušta veću voltažu prilikom otključavanja, a manju prilikom zaključavanja.

```
void Box::unlock() const {
    digitalWrite(_lockPin, HIGH);
}

void Box::lock() const {
    digitalWrite(_lockPin, LOW);
}
```

5. Zaključak

Ovaj rad je praktična izrada digitaliziranog sustava kontrole pristupa upotrebnom kreditnih kartica bez narušavanja privatnosti korisnika - osobni podaci ne dolaze u doticaj sa sustavom. Upotrebom web platforme, modernih programskih jezika, radnih okvira, biblioteka i oblaka razvijena je aplikacija za kontrolu pristupa koja omogućava administratorima sustava definiranje popisa kreditnih kartica koje imaju pristup sefu. Sef se sastoji od nekoliko elektroničkih komponenti i upravljačkog softvera koji koristi web aplikaciju pri donošenju odluke treba li sef otključati ili ne.

Implementacijom sustava moguće je odgovoriti na pitanja postavljena u uvodnom djelu rada:

- Je li kontrolor prisutan?

  S obzirom da su komponente integrirane u sef, da, kontrolor je prisutan.

- Jesu li uvjeti stvarno ispunjeni (zloupotreba ovlasti ili ljudska greška)?

  Pod pretpostavkom da je administrator definirao kartice korisnika koji smiju pristupiti sefu, da, uspješna autorizacija podrazumijeva pravilno ispunjene uvjete.

- Jesu li podaci korisnika ispravni (npr. ime i prezime)?

  Podaci o korisnicima sustavu nisu dostupni tako da, ne, ispravnost podataka nije moguće utvrditi.

- Jesu li meta podaci ispravni (npr. datum i vrijeme)?

  Pod pretpostavkom da nema vanjskog utjecaja (npr. greška u javnom NTP serveru, pokvaren interni sat računala), da, meta podaci su ispravni.

Drugi i četvrti stavak treba uzeti sa zadrškom jer se baziraju na pretpostavkama. Osim navedenih pretpostavki, također je pretpostavka da programer nije napravio logičke greške tijekom razvoja. Računalne programe nije moguće dokazati, ali ih je moguće promatrati i testirati. Pravovaljanim automatiziranim testiranjem i promatranjem (eng. *automated testing and observability*) podiže se pouzdanost pravilnoga rada sustava. Navedeni problemi ne bi trebali biti razlog odbacivanja digitaliziranog sustava kontrole pristupa jer pravilnim dizajniranjem i testiranjem nadmašuje ručni sustav kontrole pristupa.

Popis literature

[1] IBM, *What is IT Infrastructure?* Preuzeto 30. kolovoza 2021. adresa: https://www.ibm.com/topics/infrastructure.

[2] Docker, *What is a Container?* Preuzeto 31. kolovoza 2021. adresa: https://www.docker.com/resources/what-container.

[3] Kubernetes, *Kubernetes*, Preuzeto 1. rujna 2021. adresa: https://kubernetes.io.

[4] itemis, *What is a state machine?* Preuzeto 14. rujna 2021. adresa: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_state_machines.

[5] WEMOS, *D32 — WEMOS documentation*, Preuzeto 15. rujna 2021. adresa: https://www.wemos.cc/en/latest/d32/d32.html.

[6] e-radionica, *RFID reader MFRC-522 with RFID card*, Preuzeto 15. rujna 2021. adresa: https://e-radionica.com/en/rfid-reader-mfrc-522-with-rfid-card.html.

[7] SparkFun, *Magnetic Door Switch Set - COM-13247 - SparkFun Electronics*, Preuzeto 15. rujna 2021. adresa: https://www.sparkfun.com/products/13247.

[8] robotistan, *5mm Transparent RGB LED*, Preuzeto 15. rujna 2021. adresa: https://www.robotistan.com/5mm-transparent-rgb-led.

[9] eBay, *DC 12V 0.5A Cabinet Door Drawer Electric Lock Assembly Solenoid Lock Silver | eBay*, Preuzeto 15. rujna 2021. adresa: https://www.ebay.co.uk/itm/DC-12V-0-5A-Cabinet-Door-Drawer-Electric-Lock-Assembly-Solenoid-Lock-Silver/401573399553?hash=item5d7fa3cc01:g:eo8AAOSwaPNb8Os-.

[10] Amazon, *3v Relay Board Power Switch*, Preuzeto 15. rujna 2021. adresa: https://www.amazon.com/Flymmy-Channel-Optocoupler-Isolation-Development/dp/B085FQR3ZX/ref=sr_1_1_sspa?dchild=1&keywords=3v+Relay+Board+Power+Switch&qid=1631697077&sr=8-1-spons&psc=1&spLa=ZW5jcnlwdGVkUXVhbGlmaWVyPUFFM0NXUFI4RjJWRzgmZW5jcnlwdGVkSWQ9QTAzNTc2NDExSEU5
=.

[11] Chipoteka, *Adapter switch. DC 3-12 V 1A, sa 8 konektora+USB*, Preuzeto 15. rujna 2021. adresa: https://www.chipoteka.hr/artikl/27404/adapter-switch-dc-3-12-v-1a-sa-8-konektorausb-mw3k10gs-8061405628.

[12] Cadence, *The Flyback Diode: Voltage Problems and Switching Solutions*, Preuzeto 15. rujna 2021. adresa: https://resources.pcb.cadence.com/blog/2019-the-flyback-diode-voltage-problems-and-switching-solutions.

[13] tzapu, *tzapu/WiFiManager: ESP8266 WiFi Connection manager with web captive portal*, Preuzeto 16. rujna 2021. adresa: https://github.com/tzapu/WiFiManager.

Popis slika

Popis tablica